

Structured programs – Arithmetic types

Basics of Programming 1



G. Horváth, A.B. Nagy, Z. Zsóka, P. Fiala, A. Vitéz

September 17, 2025

Contents

1 C language basics

- History
- The first program
- Variables
- Scanning data, inputting

2 Structured programming in C

- Sequence
- Selection control in C
- Top-test loop

■ Application

3 Other structured elements

- Another top-test loop
- Bottom-test loop
- Integer-value based selection

4 Arithmetic types of C

- Introduction
- Integers
- Real

Chapter 1

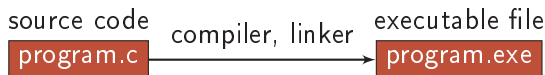
C language basics

Short history of C programming language

- 1972: Start of development at AT&T Bell Labs
Most of the UNIX kernel was created in C
- 1978: K&R C – Brian Kernigham, Dennis Ritchie:
The C Programming Language
- 1989: Standardization: ANSI X3.159-1989
- 1999: C99-standard:
new data types (complex)
international character encoding
arrays with variable sizes
...
- 2007–: C1X standard, 2011: C11 standard
C++ compatibility
multi-thread programs
...

Main features of C

- Compiled language



- "small language": few (10) instructions, a lot of (>50) operators
- concise syntax ("zipped")
 - hard to read (must pay attention)
 - easy to make a mistake
 - hard to find a mistake
- it gives a code that can be optimized efficiently and runs fast
- easy to implement for different platforms

The first C program

■ The source code of the minimum-program

```
1 /* first.c -- The first program */
2
3 int main()
4 {
5     return 0;
6 }
```

[link](#)

- The program starts, and after that it ends (finishes its run)
- between `/*` and `*/` there are comments: messages for the programmer
- `int main()` – All C programs starts like this
 - `int Main()` – and not like this. C is "case sensitive"
- `{ }` – block, it encloses the program body
- `return 0;` – It marks the end of the program

The first C program

- ...that actually does something

```
1  /* Helloworld.c -- My first program */
2  #include <stdio.h> /* needed for printf */
3
4  /* The main program */
5  int main()
6  {
7      printf("Hello world!\n"); /* Printing */
8      return 0;
9  }
```

[link](#)

- After compiling and running it gives the following output:

```
Hello world!
```

- `#include` – to insert other C program parts
- `printf` – printing, `\n` – new line (line feed)

A more complicated one

■ Instructions in a sequence

```
1  /* football.c -- football fans */
2  #include <stdio.h>
3  int main()
4  {
5      printf("Are you"); /* no new line here */
6      printf(" blind?\n"); /* here is new line */
7      printf("Go Bayern, go!");
8      return 0;
9  }
```

[link](#)

```
Are you blind?
Go Bayern, go!
```


Printing the value of a variable

```
1 #include <stdio.h>
2 int main()
3 {
4     int n;           /* declaring an integer var., called n */
5     n = 2;           /* n <- 2 assignement of value */
6     printf("The value is: %d\n", n); /* printing */
7     n = -5;          /* n <- -5 assignement of value */
8     printf("The value is: %d\n", n); /* printing */
9     return 0;
10 }
```

[link](#)

```
The value is: 2
```

```
The value is: -5
```

- `int n` – declaration of variable.
`int` (integer number) is the type, `n` is the identifier
- `n = 2` – assignement of value, variable `n` takes value of expression "2"

... continued

```
1 #include <stdio.h>
2 int main()
3 {
4     int n;           /* declaring an integer var., called n */
5     n = 2;           /* n <- 2 assignment of value */
6     printf("The value is: %d\n", n); /* printing */
7     n = -5;          /* n <- -5 assignment of value */
8     printf("The value is: %d\n", n); /* printing */
9     return 0;
10 }
```

[link](#)

- `printf(<format>, <what>)` –
printing the value of expression `<what>` in the given `<format>`
format
 - `%d` – decimal (decimal number system)

Block and declaration

Structure of the block

```
{  
    <declarations>  
    <instructions>  
}
```

```
1 {  
2     /* declarations */  
3     int n;  
4  
5     /* instructions */  
6     n = 2;  
7     printf("%d\n", n);  
8 }
```

Block and declaration

Structure of declaration

```
<type name> <identifier> [ = <initial value> ]opt;
```

```
1 int n; /* not initialized */  
2 int number_of_dogs = 2; /* initialized */
```

- value of `n` is garbage from memory at the beginning
- value of `number_of_dogs` is 2 at the beginning

Inputting data

```
1  /* square.c -- square of a number */
2  #include <stdio.h>
3  int main()
4  {
5      int num;          /* declaring an integer var. */
6      printf("Please give an integer value: "); /* info */
7      scanf("%d", &num);          /* inputting */
8      /* printing the value of 2 expressions */
9      printf("The square of %d is: %d\n", num, num*num);
10     return 0;
11 }
```

[link](#)

```
Please give an integer value: 8
The square of 8 is: 64
```

- `scanf(<format>, &<where to>)` –
Inputting (scanning) data in `<format>` format and putting it
into `<where to>` variable

Chapter 2

Structured programming in C

Sequence in C

Forming a sequence is listing instructions one after each other

```
1 /* football.c -- football fans */
2 #include <stdio.h>
3 int main()
4 {
5     printf("Are you"); /* no new line here */
6     printf(" blind?\n"); /* here is new line */
7     printf("Go Bayern, go!");
8     return 0;
9 }
```

[link](#)

Are you blind?

Go Bayern, go!

Selection control in C – the if statement

Let's write a program, that decides if the inputted integer number is small (< 10) or big (≥ 10)!

OUT: info	
IN: x	
x < 10	
OUT:small	OUT: big

```

Let x be an integer
OUT: info
IN: x
IF x < 10
    OUT: small
OTHERWISE
    OUT: big
  
```

```

1  #include <stdio.h>
2  int main()
3  {
4      int x;
5      printf("Please enter a number: ");
6      scanf("%d", &x);
7      if (x < 10)
8          /* condition */
9          printf("small"); /*true branch*/
10         else
11             printf("big"); /*false branch*/
12         return 0;
13     }
  
```

[link](#)

```

Please give an integer number: 5
small
  
```


Selection control – the if statement

Syntax of the if statement

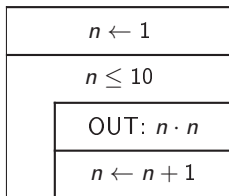
```
if (<condition expression>) <statement if true>  
[ else <statement if false> ]opt
```

```
1 if (x < 10)           /* condition */  
2   printf("small"); /* true branch */  
3 else  
4   printf("big");  /* false branch */
```

```
1 if (a < 0)           /* creating absolute value */  
2   a = -a;  
3 /* no false branch */
```

Top-test loop in C – the while statement

Let's print the square of the integer numbers between 1 and 10!



Let n be an integer

$n \leftarrow 1$

WHILE $n \leq 10$

OUT: $n \cdot n$

$n \leftarrow n + 1$

```

1  #include <stdio.h>
2  int main()
3  {
4      int n;
5      n = 1; /* initialization */
6      while (n <= 10) /* condition */
7      {
8          printf("%d ", n*n); /* printing */
9          n = n+1;
10         /* increment */
11     }
12     return 0;

```

[link](#)

1 4 9 16 25 36 49 64 81 100

Top-testing loop – the while statement

Syntax of the while statement

```
while (<condition expression>) <instruction>
```

- If <instruction> is a sequence, we enclose it in a {block}:

```
1 while (n <= 10)
2 {
3     printf("%d ", n*n);
4     n = n+1;
5 }
```

- In language C an instruction always can be replaced with a block.

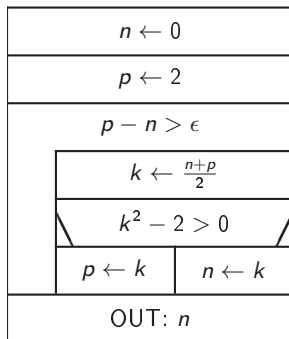
A complex application

- By using sequence, loop and selection, we can construct everything!
- We know enough to construct the algorithm of finding the zeros in C!
- A new element: a type for storing real numbers is called **double** type (to be learned later)

```
1 double a;           /* the real number */  
2 a = 2.0;           /* assignement of value */  
3 printf("%f", a);    /* printing */
```

Finding zero of a function

We are searching the zeros of function $f(x) = x^2 - 2$, between points $n = 0$ and $p = 2$, with $\epsilon = 0,001$ accuracy.



```

1  #include <stdio.h>
2
3  int main()
4  {
5      double n = 0.0, p = 2.0;
6      while (p-n > 0.001)
7      {
8          double k = (n+p)/2.0;
9          if (k*k-2.0 > 0.0)
10             p = k;
11          else
12             n = k;
13      }
14      printf("The zero is: %f", n);
15
16      return 0;
17  }
```

[link](#)

Chapter 3

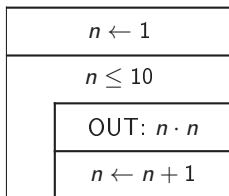
Other structured elements

Elements of structured programs

- We have seen that the structured elements we had learned so far are enough for everything.
- Only for a higher comfort, we introduce new elements, that of course origin from the earlier ones.

Top-test loop in C – the for statement

Let's print the square of the integer numbers between 1 and 10!



```
Let n be an integer
n ← 1
WHILE n <= 10
  OUT: n*n
  n ← n+1
```

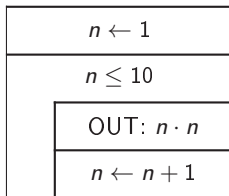
Because the structure of

- Initializations
- As long as Condition is TRUE
 - Operation
 - Increment

is very common in programming, we simplify its application with a new statement.

Top-test loop in C – the for statement

Let's print the square of the integer numbers between 1 and 10!



Let n be integer
 from $n=1$, WHILE $n \leq 10$, one-by-one
 OUT: $n \cdot n$

```

1  #include <stdio.h>
2  int main()
3  {
4      int n;
5      for (n = 1; n <= 10; n = n+1)
6          printf("%d ", n*n);
7      return 0;
8  }
```

[link](#)

1 4 9 16 25 36 49 64 81 100

Top-test loop in C – the for statement

Syntax of the for statement

```
for (<init exp>; <cond exp>; <post-op exp>)  
<instruction>
```

```
1 for (n = 1; n <= 10; n = n+1)  
2   printf("%d ", n*n);
```

- Post-operation is performed after execution of the instruction.

n: 11

1 4 9 16 25 36 49 64 81 100

Multiplication table

Let's print the 10 · 10 multiplication table!

- We have to print 10 rows (row = 1, 2, 3, ...10)
- In every row
 - we print into 10 columns (col = 1, 2, 3, ...10)
 - In every column
 - We print the value of row*col
- After this we have to start a new line

```
1 int row;
2 for (row = 1; row <= 10; row=row+1)
3 {
4     int col;        /* declaration at beginning of block */
5     for (col = 1; col <= 10; col=col+1)
6         printf("%4d", row*col); /* printing with size 4 */
7     printf("\n"); /* this is not inside the for */
8 }
```

[link](#)

Multiplication table

- It might be advantageous to enclose in a block even one single instruction, because it might make the code more understandable!

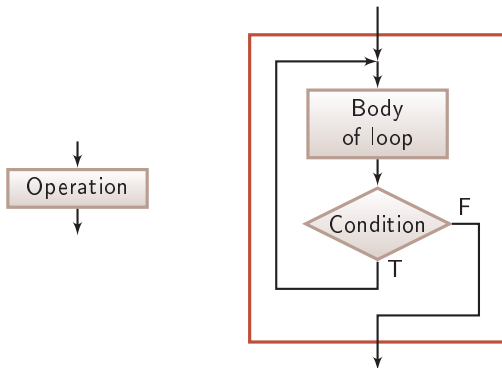
```
1  int row;  
2  for (row = 1; row <= 10; row=row+1)  
3  {  
4      int col;      /* declaration at beginning of block */  
5      for (col = 1; col <= 10; col=col+1)  
6      {  
7          printf("%4d", row*col); /* printing with size 4 */  
8      }  
9      printf("\n");  
10 }
```

[link](#)

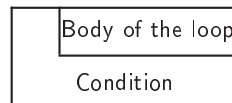
Elements of structured programs

Bottom-test loop

Repetition of an operation as long as a condition is true.

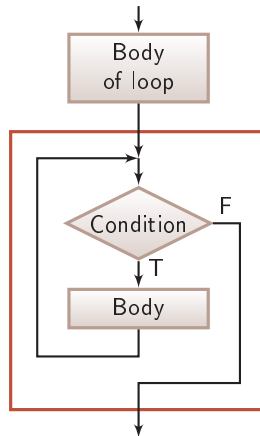
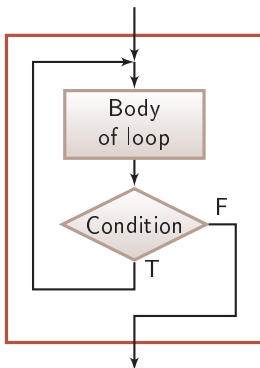


```
REPEAT
  Body of loop
WHILE Condition
```



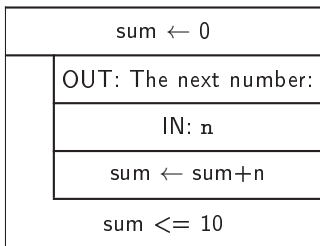
Elements of structured programs

- It can be traced back to sequence and a top-test loop



Bottom-test loop – the do statement

Let's read positive integer numbers! We stop if the sum of the numbers is larger than 10.



```

sum  $\leftarrow$  0
REPEAT
  OUT: Info
  IN: n
  sum  $\leftarrow$  sum+n
WHILE sum  $\leq$  10
  
```

```

1  #include <stdio.h>
2  int main()
3  {
4      int sum = 0, n;
5      do
6      {
7          printf("The next number: ");
8          scanf("%d", &n);
9          sum = sum+n;
10     }
11     while (sum <= 10);
12     return 0;
13 }
  
```

[link](#)

Bottom-test loop – the do statement

Syntax of the do statement

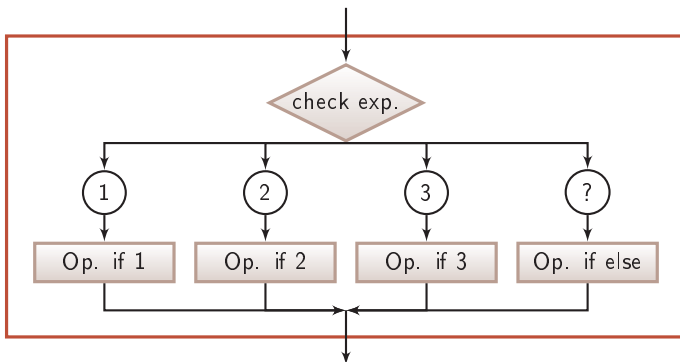
```
do <instruction> while (<condition expression>);
```

```
1 do
2 {
3     printf("The next number: ");
4     scanf("%d", &n);
5     sum = sum+n;
6 }
7 while (sum <= 10);
```


Elements of structured programs

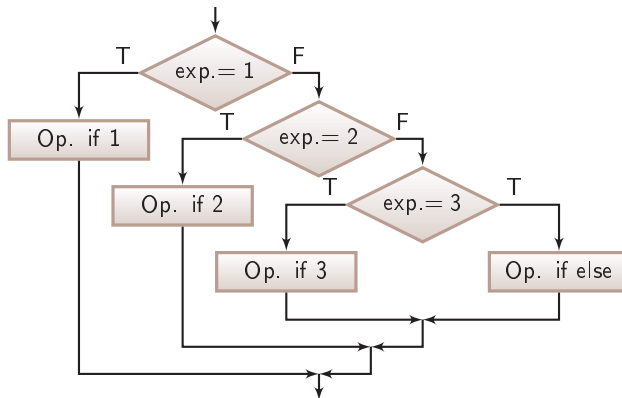
Integer-value based selection

Execution of operations depending on the value of an integer expression



Elements of structured programs

- It can be constructed as nested selections



Integer-value based selection – the switch statement

- Let's assign (connect) written evaluations to grades given in numbers!

OUT: info					
IN: n					
$n = ?$					
1	2	3	4	5	other
OUT: failed	OUT: poor	OUT: average	OUT: good	OUT: perfect	OUT: something wrong

Integer-value based selection – the switch statement

- Let's assign (connect) written evaluations to grades given in numbers!

```
1 #include <stdio.h>
2 int main() {
3     int n;
4     printf("Please enter the grade: ");
5     scanf("%d", &n);
6     switch (n)
7     {
8         case 1: printf("failed"); break;
9         case 2: printf("poor"); break;
10        case 3: printf("average"); break;
11        case 4: printf("good"); break;
12        case 5: printf("perfect"); break;
13        default: printf("something wrong");
14    }
15    return 0;
16 }
```

[link](#)

Integer-value based selection – the switch statement

Syntax of the switch statement

```
switch(<integer expression>) {  
    case <constant exp1>: <instruction 1>  
    [case <constant exp2>: <instruction 2> ...]_opt  
    [default: <default instruction> ]_opt  
}
```

```
1  switch (n)  
2  {  
3      case 1: printf("failed"); break;  
4      case 2: printf("poor"); break;  
5      case 3: printf("average"); break;  
6      case 4: printf("good"); break;  
7      case 5: printf("perfect"); break;  
8      default: printf("something wrong");  
9  }
```

Integer-value based selection – the switch statement

- The `break` instructions are not part of the syntax. If we omit them, the `switch` will remain syntactically correct, but it will not provide the same result as before:

```
1 switch (n)
2 {
3     case 1: printf("failed");
4     case 2: printf("poor");
5     case 3: printf("average");
6     case 4: printf("good");
7     case 5: printf("perfect");
8     default: printf("something wrong");
9 }
```

[link](#)

```
Please enter the grade: 2
pooraveragegoodperfectsomething wrong
```

Integer-value based selection – the switch statement

- The constant expressions are only entry points, and from this point on, all instructions are executed until the first **break** or until the end of the block:

```
1  switch (n)
2  {
3      case 1: printf("failed"); break;
4      case 2:
5      case 3:
6      case 4:
7      case 5: printf("passed"); break;
8      default: printf("something wrong");
9  }
```

[link](#)

```
Please enter the grade: 2
passed
```

Chapter 4

Arithmetic types of C

Types – Introduction

Type is

- Set of values
- Operations
- Representation

In a real computer – the set of values is limited

- We can not represent arbitrary large numbers
- We can not represent numbers with arbitrary accuracy
 $\pi \neq 3.141592654$
- We must know the limits of what can be represented, in order to store our data
 - without any loss of information or
 - with an acceptable level of information loss, without wasting memory

Types of C language

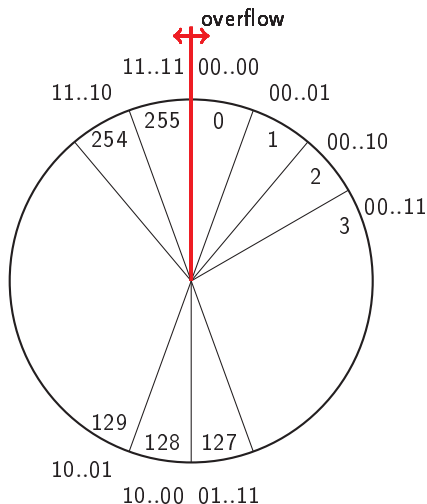
- void
- scalar
 - arithmetic
 - integer: integer, character, enumerated
 - floating-point
 - pointer
- function
- union
- compound
 - array
 - structure
- Today we will learn about them

Binary representation of integers

- Binary representation of unsigned integers stored in 8 bits

dec	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	hex
0	0	0	0	0	0	0	0	0	0x00
1	0	0	0	0	0	0	0	1	0x01
2	0	0	0	0	0	0	1	0	0x02
3	0	0	0	0	0	0	1	1	0x03
⋮	⋮							⋮	⋮
127	0	1	1	1	1	1	1	1	0x7F
128	1	0	0	0	0	0	0	0	0x80
129	1	0	0	0	0	0	0	1	0x81
⋮	⋮							⋮	⋮
253	1	1	1	1	1	1	0	1	0xFD
254	1	1	1	1	1	1	1	0	0xFE
255	1	1	1	1	1	1	1	1	0xFF

The overflow



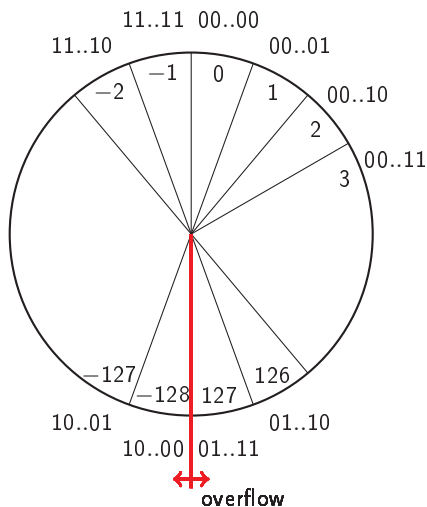
- In case of unsigned integers stored in 8 bits
 - $255+1 = 0$
 - $255+2 = 1$
 - $2-3 = 255$
- "modulo 256 arithmetic"
 - We always see the remainder of the result divided by 256

Two's complement representation of integers

- Two's complement representation of signed integers stored in 8 bits

dec	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	hex
0	0	0	0	0	0	0	0	0	0x00
1	0	0	0	0	0	0	0	1	0x01
2	0	0	0	0	0	0	1	0	0x02
3	0	0	0	0	0	0	1	1	0x03
⋮	⋮							⋮	⋮
127	0	1	1	1	1	1	1	1	0x7F
-128	1	0	0	0	0	0	0	0	0x80
-127	1	0	0	0	0	0	0	1	0x81
⋮	⋮							⋮	⋮
-3	1	1	1	1	1	1	0	1	0xFD
-2	1	1	1	1	1	1	1	0	0xFE
-1	1	1	1	1	1	1	1	1	0xFF

The overflow



- In case of signed integers stored in 8 bits

- $127+1 = -128$

- $127+2 = -127$

- $-127-3 = 126$

- on the other hand

- $2-3 = -1$

Integer types in C

type	bit ¹	<limits.h>		printf
signed char	8	CHAR_MIN	CHAR_MAX	%hhd ²
unsigned char	8	0	UCHAR_MAX	%hhu ²
signed short int	16	SHRT_MIN	SHRT_MAX	%hd
unsigned short int	16	0	USHRT_MAX	%hu
signed int	32	INT_MIN	INT_MAX	%d
unsigned int	32	0	UINT_MAX	%u
signed long int	32	LONG_MIN	LONG_MAX	%ld
unsigned long int	32	0	ULONG_MAX	%lu
signed long long int ²	64	LLONG_MIN	LLONG_MAX	%lld
unsigned long long int ²	64	0	ULLONG_MAX	%llu

¹Typical values, the standard only determines the minimum

²since the C99 standard

Declaration of integers

■ Defaults

- The `signed` sign-specifier can be omitted

```
1  int i;           /* signed int */
2  long int l;      /* signed long int */
```

- If there is sign- or length-modifier, the `int` can be omitted.

```
1  unsigned u;      /* unsigned int */
2  short s;         /* signed short int */
```


Integer types

- An example on how to use the previous table: a program that runs for a very long time³

```
1 #include <limits.h> /* for integer limits */
2 #include <stdio.h> /* for printf */
3
4 int main(void)
5 { /* almost all long long int */
6     long long i;
7
8     for (i = LLONG_MIN; i < LLONG_MAX; i = i+1)
9         printf("%lld\n", i);
10
11     return 0;
12 }
```

[link](#)

³provided that `long long int` is 64 bit long, the program runs for 585 000 years if the computer prints 1 million numbers per second

Integer constants

■ Specifying integer constants

```
1 int i1=0, i2=123, i4=-33;           /* decimal */
2 int o1=012, o2=01234567;          /* octal */
3 int h1=0x1a, h2=0x7fff, h3=0xAa1B /* hexadecimal */
4
5 long l1=0x1a1, l2=-33L;            /* l or L */
6
7 unsigned u1=33u, u2=45U;           /* u or U */
8 unsigned long ul1=33uL, ul2=1231U; /* l and u */
```

- If neither u or l is specified, the first type that is big enough is taken:

- 1 int
- 2 unsigned int – in case of hexa and octal constants
- 3 long
- 4 unsigned long

Why do we need to know the limits of number representations?



Let's determine the following value!

$$\binom{15}{12} = \frac{15!}{12! \cdot (15 - 12)!}$$

(What is the number of possibilities of selecting 12 out of 15 different chocolates?)

- The value of the numerator is $15! = 1\,307\,674\,368\,000$
- The value of the denominator is $12! \cdot 3! = 2\,874\,009\,600$
- None of them can be represented as a 32 bits `int`!
- But with simplifying the expression

$$\frac{15 \cdot 14 \cdot 13}{3 \cdot 2 \cdot 1} = \frac{2730}{6} = 455$$

all parts can be calculated without any problem, even on 12 bits.

Floating-point types

■ Normal form

$$\begin{aligned} 23.2457 &= (-1)^0 \cdot 2.3245700 \cdot 10^{+001} \\ -0.001822326 &= (-1)^1 \cdot 1.8223260 \cdot 10^{-003} \end{aligned}$$

Representation of the normal form

■ Floating-point fractional = sign bit + mantissa + exponent

- 1 sign bit: 0–positive, 1–negative
- 2 mantissa: unsigned integer (without the decimal comma), because of normalization, the first digit is ≥ 1
- 3 exponent (or order, characteristic): signed integer

Floating-point types

■ Binary normal form

$$5.0 = 1.25 \cdot 4 = (-1)^0 \cdot 1.0100_b \cdot 2^{010_b}$$

0	0100	010
---	------	-----

Representation of binary normal form

- Floating-point fractional = **sign bit** + **mantissa** + **exponent**
 - 1 **sign bit**: 0–positive, 1–negative
 - 2 **mantissa**: unsigned integer (without the **binary comma**), because of normalization, the first digit is = 1, so we don't store it⁴.
 - 3 **exponent**: signed integer

⁴the leading bit is implicit

Floating-point types in C

■ Floating-point types of C

type	typical values			printf/scanf
	bits	mantissa	exponent	
<code>float</code>	32 bits	23 bits	8 bits	<code>%f</code>
<code>double</code>	64 bits	52 bits	11 bits	<code>%f/%lf</code>
<code>long double</code>	128 bits	112 bits	15 bits	<code>%Lf</code>

■ Floating-point constants

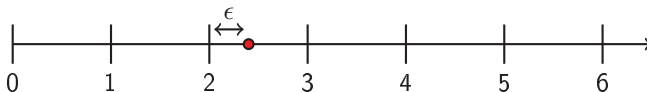
```

1 float      f1=12.3f , f2=12.F , f3=.5f , f4=1.2e-3F ;
2 double     d1=12.3 , d2=12. , d3=.5 , d4=1.2e-3 ;
3 long double l1=12.31 , l2=12.L , l3=.51 , l4=1.2e-3L ;

```

■ In C we use decimal point and not a comma!

Representation accuracy of integer types

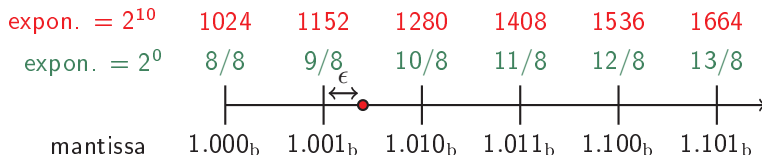


Absolute accuracy of number representation

It is the maximal ϵ error of representing an arbitrary real number with the closest integer

- The absolute accuracy of representing with integer types is 0.5

Representation accuracy of floating-point numbers



- in this example
 - The (absolute) representation accuracy of the mantissa is $1/16$
 - If the exponent is 2^0 , the representation accuracy is $1/16$
 - If the exponent is 2^{10} , the representation accuracy is $2^{10}/16 = 64$
- There is no absolute, only relative accuracy, that is, in this present case, 3 bits.

Consequences of finite number representation

- As the floating-point number representation is not accurate, we **must not** check the equality of results of operations!

$$\frac{22}{7} + \frac{3}{7} \neq \frac{25}{7}$$

instead

$$\left| \frac{22}{7} + \frac{3}{7} - \frac{25}{7} \right| < \varepsilon$$

- The exponent will magnify the rounding error of the finite long mantissa, thus the large numbers are much less accurate than small numbers. The errors of the large numbers can "eat up" the small ones:

$$A + a - A \neq a$$

Consequences of the binary representation of numbers

- A decimal finite number might not be finite in binary form, eg.:

$$0,1_d = 0,000\overline{11}_b$$

- How many times will be this cycle repeated?

```
1 double d;  
2 for (d = 0.0; d < 1.0; d = d+0.1) /* 10? 11? */  
3 {  
4     ...  
5 }
```

- The good solution is:

```
1 double d;  
2 double eps = 1e-3; /* what is the right eps for here? */  
3 for (d = 0.0; d < 1.0-eps; d = d+0.1) /* 10 times */  
4 {  
5     ...  
6 }
```

Thank you for your attention.